Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 16 pages

Development of Safety-Critical Reconfigurable Hardware with Esterel

Jerker Hammarberg Simin Nadjm-Tehrani [1]

Department of Computer and Information Science
Linköping University
Linköping, Sweden

Abstract

Demands for higher flexibility in aerospace applications has led to increasing deployment of FPGAs. Clearly, analysis of safety-related properties of such components is essential for their use in safety-critical subsystems. The contributions of this paper are twofold. First, we illustrate a development process, using a language with formal semantics (Esterel) for design, formal verification of high-level design and automatic code generation down to VHDL. We argue that this process reduces the likelihood of systematic (permanent) faults in the design, and still produces VHDL code that is of acceptable quality (size of FPGA, delay). Secondly, we show how the design model can be modularly extended with fault models that represent random faults (e.g. radiation) leading to bit flips in the component under design (resembling FMEA), and transient or permanent faults in the rest of the environment (corrupting inputs to the component or jeopardising the effect of output signals that control the environment). The set-up is then used to formally determine which (single or multiple) fault modes cause violation of the top-level safety-related property, much in the spirit of fault-tree analyses. An aerospace hydraulic monitoring system is used to illustrate the results.

Key words: Safety Analysis, Formal Verification, FPGA, Esterel

1 Introduction

The drive for producing faster, cheaper, and better systems in the last decade has made the need for reuse of models in development cycles of safety-critical systems a reality. Contrary to current life cycle processes in which several teams use various partially overlapping models and documents in almost independent activities, one now looks for ways to reuse the same component (model) in different activities using different views. An important example

[1] Email: simin@ida.liu.se

of such component-based reuse is when a formal design model captures the functions of a component to be integrated into a system. In this paper we propose that the same (mathematical) *functional* model is used when analysing the *safety-related* analysis at the system level. This opens up for reuse of the component both in different parallel processes in the same development cycle, and with respect to reuse in future generations.

This paper argues for such a reuse methodology in the context of a specific example: design of reconfigurable and safety-critical hardware. The results thus build on languages and tools that promote flexible and dependable hardware. However, the generic message of the paper is the merger between the functional and safety-related analyses that are by no means language or application-dependent.

The combination of flexibility and efficiency requirements in development of digital components has made the use of Field Programmable Gate Arrays (FPGAs) prevalent, not only in space electronics [11], but also in more traditional avionic systems. In a recent study of an Air Intercept missile system, FPGAs were used at design stage to provide flexibility, but were carried over to the tactical system due to cost considerations as compared with the more prohibitive Application Specific Integrated Circuits (ASICs) [9]. This trend necessitates development of techniques for evaluation of the safety risks associated with FPGAs.

When considering traditional safety evaluation techniques, FPGAs lie somewhere in between hardware and software. Although traditional quantitative safety/reliability analysis favours the use of hardware in safety-critical applications, recent results show that this may be a changing reality. Shivakumar et. al. estimate that soft error rates per chip (bit flips in hardware as a result of cosmic radiation) of logic circuits will increase nine orders of magnitude in a ten year perspective by 2011, at which point they will be comparable to the soft error rate per chip of unprotected memory units [16].

The above considerations motivate a serious study of system development processes that facilitate efficient integration of FPGAs and other software or hardware components in systems design, with improved levels of safety. In this paper we study the integration of FPGA designs in safety-critical aerospace applications by illustrating two aspects of fault-management. First, we consider techniques that help the developer to remove or prevent systematic (permanent) design faults. Second, we put forward an analysis method that helps the system safety engineer to concentrate on combinations of external and random faults that should be the focus of the fault tolerance and containment techniques. The suggested method combines elements of analysis normally performed to concentrate on faults that lead to system level hazards, much in the spirit of fault-tree analysis (FTA) and failure modes and effects analysis (FMEA) techniques [8]. The novelty of our approach is the combination of this type of safety-related analysis with the model-driven development of a design, early formal verification, and automatic code generation. The contri-

butions of the paper are therefore not in development of core techniques, but in bringing together a number of ingredients that have so far existed as isolated activities, by providing a systematic approach for introduction of engineering knowledge in a formal development process. This approach is illustrated on a real aerospace case study, and the resulting design justified in terms of efficiency and performance characteristics.

The remainder of the paper is structured as follows: Section 2 presents the system that was used for the case study. Section 3 introduces some of the necessary concepts and techniques involved in the development and safety analysis of the system. Section 4 describes our approach to a combined development of reusable design models and safety-related analyses. Finally, section 5 summarises the paper and presents the future work.

2 Aerospace Application

The ideas presented in this paper were motivated by the needs for safety analysis of a subsystem inside the JAS 39 Gripen multi-role aircraft, obtained from Saab Aerospace AB in Linköping, Sweden. The purpose of the system is to detect and stop leakages in the two hydraulic systems, which feed the moving parts, including the flight control surfaces, with mechanical power. Leakages in the hydraulic systems could in the worst case result in such low hydraulic pressure that the airplane becomes uncontrollable. To avoid this, some of the branching oil pipes are protected by shut-off valves. These valves can be used to isolate a branch in which a leakage has been detected. Then, although the leaking branch will no longer function, the other branches will still keep the pressure and be able to supply the moving parts with power.

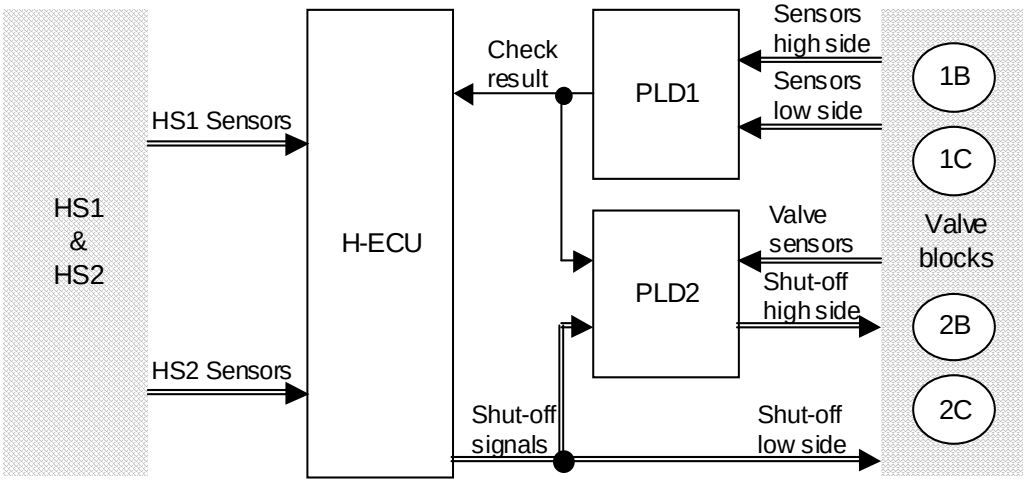

Fig. 1. Hydraulic leakage detection subsystem. White boxes indicate electronic components and patterned boxes indicate physical parts. Arrows indicate signal flows; double arrows are collections of several signals.

The reading of oil level sensors and the controlling of the four shut-off valves are handled by three electronic components, as depicted in figure 1. The H-ECU is a software component that continually reads the oil reservoir levels of the two hydraulic systems, and determines which shut-off valve to close accordingly. However, it would be very dangerous if some fault caused more than one valve to close at the same time — this could result in the locking of the flight control surfaces. For this reason, two programmable logic devices, here called PLD1 and PLD2, continually read the signals to and the statuses of the valves, and if the readings indicate closing of more than one valve, they will disallow further closing. Thus, PLD1 and PLD2 add fault tolerance to the shut-off subsystem. PLD2 only accepts a request from the H-ECU of closing a particular valve if the check, which is partly done in PLD1, indicates that everything is OK. A valve will only close when both the low side signal, which is the shut-off signal directly from the H-ECU, and the high side signal, which is the checked signal from PLD2, are present.

Design for fault tolerance is an inherent part of satisfying safety requirements for a system. Although quantification of dependability using selected metrics is well-studied in electromechanical systems, satisfaction of safety requirements in presence of software or software-like hardware is still a major challenge. In this particular example, one wants to verify that no single fault, be it in the components, in the wires or in the valves, can cause more than one valve to close at the same time. Moreover, the safety engineer is interested in combinations of faults that might lead to violation of top safety requirements. In the following, we will show how a design model of the system can be used to formally verify that it is tolerant to single faults. The proofs also pinpoint the significant combinations of double faults. Since model-based development down to code generation and formal verification is supported by the Esterel language and tools, these will be used to illustrate the ideas.

3 Background

This section introduces some concepts and earlier work that will be used for the analysis described in section 4. An introduction to the properties of Esterel is followed by a short description of fault tree analysis and how formal analysis can be applied in this context.

3.1 Esterel

Esterel [1] is a synchronous language with formal semantics, making it ideal for analyses with formal methods. In addition, a large subset of Esterel can be directly compiled to synthesizable VHDL or Verilog, and thus synthesized to FPGA configuration data without any need for manual rewriting. This means that if systems are designed in Esterel, efficient formal verification is available directly on the actual design code.

Esterel is tailored for designing reactive systems [13], which continually read inputs and produce outputs. In Esterel, time is modelled as a discrete sequence of instants. Each instant, new outputs are computed from the inputs and from the internal state (control state, latched signals and variables) according to the imperative statements of the Esterel program. Figure 2 shows some of the Esterel code for PLD2 in the aerospace example. Every instant, the code inside the `loop...each tick` construct is executed. The output signal *ShutOff_1B* will be emitted only if there is an incoming request to close valve 1B, all input seems to be OK and the *CheckOK* signal was present at the previous instant.

```
module PLD2:
   input CheckOK;
   input ShutOffRequest_1B;
   % more inputs
   output ShutOff_1B;
   % more outputs
   loop
      signal InputNotOK in
         % some code emitting InputNotOK if something is wrong with the input
         present ShutOffRequest_1B and not InputNotOK and pre(CheckOK) then
            emit ShutOff_1B
         end present
      end signal
   each tick
end module
```

Fig. 2. Part of the Esterel code for PLD2.

Our experience [5] indicate that synchronous hardware systems can be written more easily and with fewer lines of code in Esterel than in hardware description languages such as VHDL [17]. This is due to the higher level of abstraction and the suitable constructs for modelling hierarchical reactive modules. Bug count per thousand lines of code (bugs/kloc) appears to be constant over different languages [15]. Hence, we believe that the use of Esterel could contribute to system correctness and maintainability at a lower cost (i.e. development time), compared to the design methods that are commonly used in the industry today.

The main advantage with using Esterel for hardware development is, however, its tight coupling to formal methods for system analysis. Esterel compilers automatically check the design for causality loops, and the fact that Esterel is synchronous makes formal verification particularly efficient [6]. Nondeterministic programs are rejected at compilation stage, making the reliance on the output of a module at each instant possible. The commercial tool Esterel Studio [4] has two model checkers built-in — one based on binary decision diagrams (BDDs) [14] and one based on propositional satisfiability (SAT) techniques [18] [19]. Thus, many design faults should be quickly found and eliminated already at the design stage of the process.

The two model checkers bring the best of both worlds in the battle against complexity, namely state space explosion and long proofs with many open hypotheses, respectively. The tool used in our experiments was the SAT solver. It implements the Stålmarck proof technique that is based on the observation that many systems with large state spaces have short proofs [2]. The interested reader is referred to the tutorial by Sheeran and Stålmarck [19] for a full exposure to the method, since it is beyond the scope of this paper.

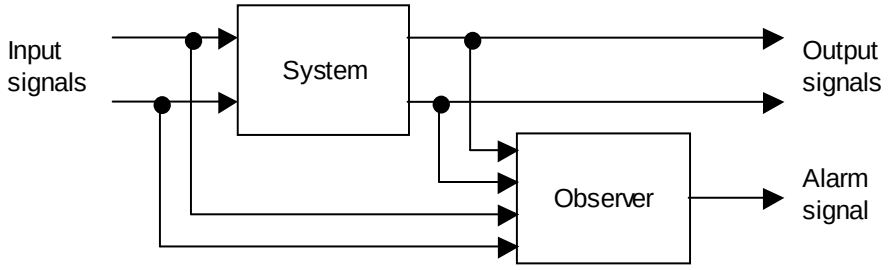

Fig. 3. A system being monitored by an observer. The arrows indicate signal flow or signal readings.

In Esterel, the safety properties [3] to prove with the model checker are formalised as synchronous observers. The *observer* is a process, also written in Esterel, that runs in parallel with the actual system and monitors its input and output signals, as depicted in figure 3. As soon as the observer finds that the property is violated, it emits an *alarm signal* (sometimes also called *bug signal*). Proving the property is thus reduced to proving that the alarm signal will never be emitted.

A justified concern is whether the generated VHDL code is efficient enough to be useful in a real industrial application. In this particular case study, the PLDs are combinatorial. Thus, the size differences are not significant. Our experiments encompassed another system [5], in which we compared an FPGA implementation synthesized from Esterel with a hand-written VHDL implementation of the same system. The Esterel implementation was approximately three times slower and occupied slightly more than two times of the logic blocks (CLBs) in the FPGA. Although more studies are needed to be able to draw any real conclusions, this indicates that VHDL is still the appropriate choice if efficiency is crucial, but in other cases, the acceptance criteria may simply be fitting into the chip. In these cases, Esterel can produce implementations of fully acceptable size and speed. We also argue that the additional cost of using a larger FPGA, should the implementation not fit, will in many cases be minor compared to the potential gains resulting from shorter development time and increased reliability. Moreover, ongoing research (see for instance [3] and [10]) aims at further improving the compilation of Esterel to hardware,

[2] In the sense of Gentzen style deductions.
[3] In the formal verification sense: non-reachability of a bad state.

and the language itself is being developed to allow for better control over the hardware resources.

Finally, it should be noted that there are other synchronous languages, e.g. Signal [12] and Lustre [7], that support formal verification. However, none of them can presently be compiled to VHDL (and thus not synthesized to an FPGA) with a commercially available compiler. This is the main reason for Esterel being chosen in our studies.

3.2 Fault tree analysis (FTA)

A traditional technique for safety analysis is to produce a *fault tree* that displays the hazardous events and their possible implicants. In a fault tree, each node represents an event, and the *top event* is the disaster that one wants to find the possible causes to. The children of an event in the tree are the events that, alone or together, may cause that event. By producing a complete fault tree, one can reveal what faults (or combinations of faults) are hazardous and must be attended to. In addition, if the probabilities for the leaf events are known, a probability for the top event can be calculated.

Consider now a fault tree for the aerospace application, where the top event is the airplane crashing. One event (internal node in the fault tree) that can single-handedly cause the top event is when two or more shut-off valves close simultaneously. To further derive the descendants of this event in the fault tree would require a complete analysis of the behaviour of the three electronic components H-ECU, PLD1 and PLD2; something that would be extremely tedious to do in separate models developed in FTA tools. Clearly, there is a need for tool support to create the fault tree automatically. Moreover, every change in the design would potentially render reconstructing the fault tree.

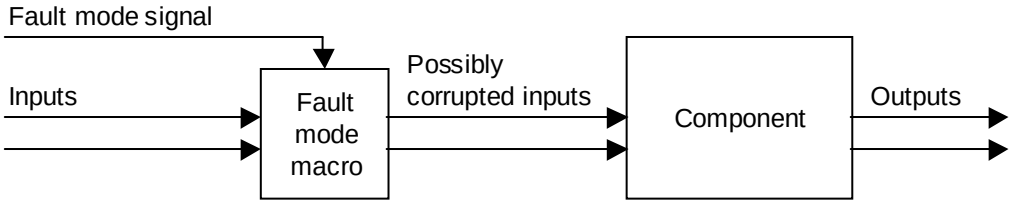

Fig. 4. Extending a component with a fault mode. Arrows indicate signal flow.

Åkerlund et al, 1999, showed how FTA-like analysis of a hardware or a software component could be performed with the help of a SAT solver [21]. The component was first modelled as a set of logic formulas relating the input and output variables. By checking that the formulas that constitute the model imply a specific property, such as "at most one shut-off valve is signalled to close simultaneously", compliance with a safety (correctness) property in the absence of external faults could be proved. This is ordinary model checking. Then, the model was extended with additional input signals representing specific fault modes in the environment that could cause corruption of the

real input to the component. A fault was modelled by a macro that modifies the input signals if the fault mode signal is present, as illustrated in figure 4. By again verifying this extended model, one could either prove that the component was tolerant to the modelled faults or see on the produced counterexamples which fault modes could cause violation of the property. The counterexamples were automatically generated by the verification tool, as a side effect of the analysis. Note that a fault tree was never built explicitly, but the setup allows the study of effects of single as well as multiple faults (including those with non-monotonic effects).

4 Analysis of fault tolerance

In this section, we combine and extend the above techniques to obtain several advantages. First, instead of low level (circuit) modelling of the design as in the Åkerlund et al study, we lift the design abstraction to a higher level. Due to maturity of Esterel tools, high-level design can now be used for automatic code generation. Without this step, our original ideas with automated FTA-like analysis are far from applicability in every day industrial settings. Secondly, we extend the classes of considered faults. In addition to faulty inputs of the component under design, we model faulty outputs generated by the component itself as a result of random failures, as well as faults in the environment that consumes the output of the component.

We illustrate that the ideas are practical and that the Esterel tool makes them immediately available already at the design stage. Hence we remove the need for creating and managing two separate models (one for design verification, and one for FTA/FMEA analysis of safety-related properties). In addition, the idea builds on a framework for combining individual components to form a verification bench that allows the study of the effects of the above fault classes.

The three steps involved in formally analysing fault tolerance will be explained in the following.

4.1 Development of verification bench

In order to verify the hardware and software components working together with physical parts of the system, it will be necessary to build an Esterel model of relevant parts of the environment. In addition to models of the physical parts of the system, we include models of all necessary wires between various components as well as wires between the components and the physical parts. Next, observers running in parallel with the components and the environment are added. This construction will be referred to as a *verification bench*, into which various components can be plugged in and analysed.

A verification bench for the aerospace application is illustrated in figure 5. It contains models of the four valves and of the wires between the components,

and it has empty slots for H-ECU, PLD1 and PLD2. The observer monitors the output of the valve models and emits an alarm signal as soon as more than one valve is closed at the same instant.

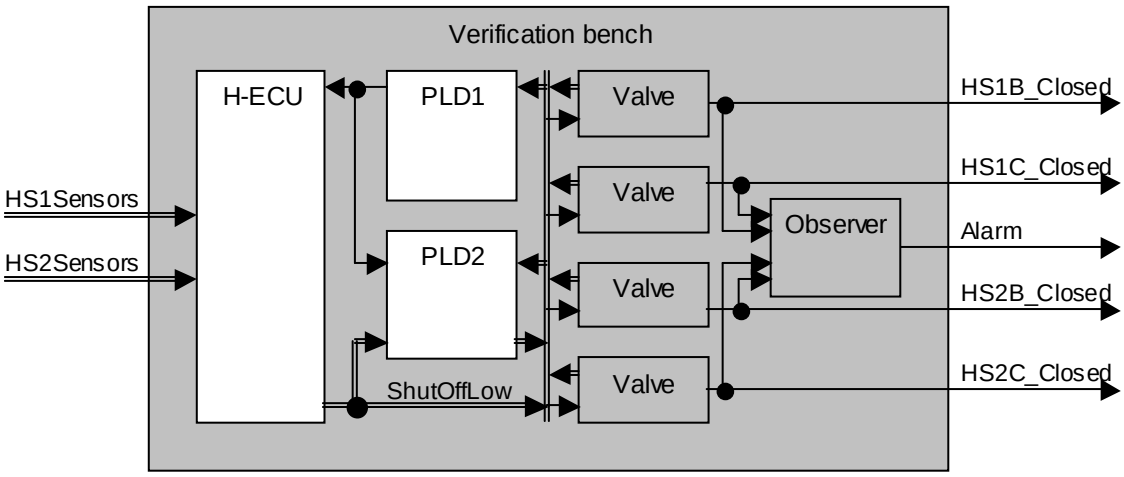

Fig. 5. Verification bench for aerospace application. Grey boxes indicate modules of the verification bench and white boxes indicate modules to be verified. Arrows indicate wires; double arrows are collections of several wires. The vertical bar in the middle is a short-hand for a set of connections.

Figure 6 shows a skeleton for the implementation of this verification bench in Esterel. All wires are modelled with local signals using the `signal` construct, and the main body consists of calls to the components (`run` keyword) and the valve models running in parallel with the observer that checks that no pair of valve-closed signals are simultaneously present. The signal renamings in the `run` construct defines how the interface of the component is connected to the local signals or to the main inputs or outputs.

Notice that the verification bench can be written independently of the components. This is useful for distributed development — code from different departments, or even different companies, can easily be plugged in and immediately checked by formal verification.

4.2 Augmenting with fault modes

The next step in the process of checking for safety-related fault tolerance is to model faults in the verification bench. Here we will show how to model malfunctions in the chips on which the components run, in our case the microprocessor and the two PLDs. Many other classes of faults, such as electrical or mechanical faults in physical parts of the system, can also be modelled in the verification bench with some creativity.

Faults in the hardware parts such as FPGAs or microprocessors on which the system components run can occur due to sudden power-down, overheating or radiation that flips some bits over inside the chip. Modelling such faults at

```
module Main:
   sensor HS1Pressure : double;
   % more main inputs
   output HS1B_Closed;
   % more main outputs
   output Alarm;
   signal
      ShutOffLow_1B;
      % more wires
   in
      run HECU [
         signal HS1Pressure / HS1Pressure;
         % more connections
      ]
      ||
      run PLD1 [...]
      ||
      run PLD2 [...]
      ||
      run Valve [...]
      ||
      % more valves
      ||
      loop
         present HS1B_Closed and HS1C_Closed then
            emit Alarm;
         end present;
         % more checks
      each tick
   end signal
end module

module Valve:
   % valve model
end module
```

Fig. 6. Skeleton code for aerospace application verification bench in Esterel.

fine granularity would be complicated, so we will opt for a coarse granularity strategy and assume that if such a fault occurs, the outputs of the component running on the chip can be anything. Besides being much simpler, this strategy also has the advantage that the component design module does not have to be altered; the malfunction can be completely modelled in the verification bench.

To induce completely arbitrary output from a component, one can add an additional block coming in between the outputs of the component and the following wires, as depicted in figure 7. This block will be referred to as a *fault switch* and can be seen as the formal verification counterpart of fault injectors used in test benches. Figure 8 shows the Esterel implementation of a fault switch following the H-ECU, and it should replace the `run HECU` call in figure 6. The idea is to let through the correct output of the component into the wires as long as the fault mode signal is absent, but when it is present, arbitrary

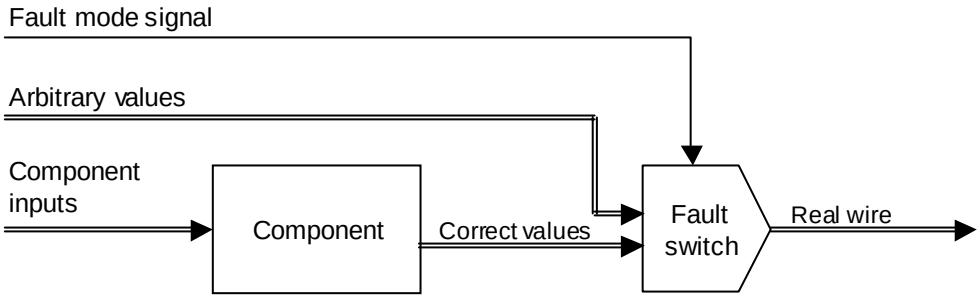

Fig. 7. Fault switch modelling component faults. Double arrows indicate possibly several signals.

```
...
run HECU [
   signal HS1Pressure / HS1Pressure;
   % more connections
   % feed output to local correct-values wire
]
||
loop
   present FaultHECU then
      % feed arbitrary values to local signals (ShutOffLow_1B etc)
   else
      % feed correct values to local signals
   end present
each tick
...
```

Fig. 8. Skeleton code for fault switch in Esterel.

values will be fed into the wires instead. These arbitrary values can be taken from additional inputs of the verification bench, since the verification tool will allow these to be anything. The component must furthermore be connected to the inputs of the fault switch instead of the wires.

In the aerospace application case, the following faults were modelled:

- Arbitrary malfunction in the H-ECU, in PLD1 or in PLD2 (e.g. due to radiation).
- Short-cut to ground on the incoming low side shut-off signal to each of the four valves.
- Short-cut to ground on the low side of each valve [4] .
- Short-cut between the low and the high side of each valve.

[4] This is not the same as grounding of the incoming low side shut-off signal, since there are electronic components in the valve that, based on the incoming signals, produce a voltage that makes the valve close. This fault models grounding of the low side of the voltage.

4.3 Fault mode verification

In Esterel Studio, plugging in the designs of the components is simply a matter of loading them into the verification bench project. The verification bench and the components together then constitute a model of the complete subsystem that can be verified with the built-in model checker.

To analyse safety-related fault tolerance, we make use of the feature in Esterel Studio to constrain some input signals. In this case the fault mode signals can be restricted to analyse different scenarios. By further testing for the observer alarm signal emission, we can check for violation of the safety property in presence of those faults. For example, if we are only interested in component malfunction to see what combinations of faulty components the system can withstand, we can constrain all other fault mode signals to be absent. If we want to find systematic design faults, that is, violations of properties in absence of external faults, we simply constrain all fault mode signals to be absent. This is probably the first thing we want to do. Single (and possibly double) faults can be found by allowing all single (and all pairs of) fault modes, constraining the other fault mode signals to be absent.

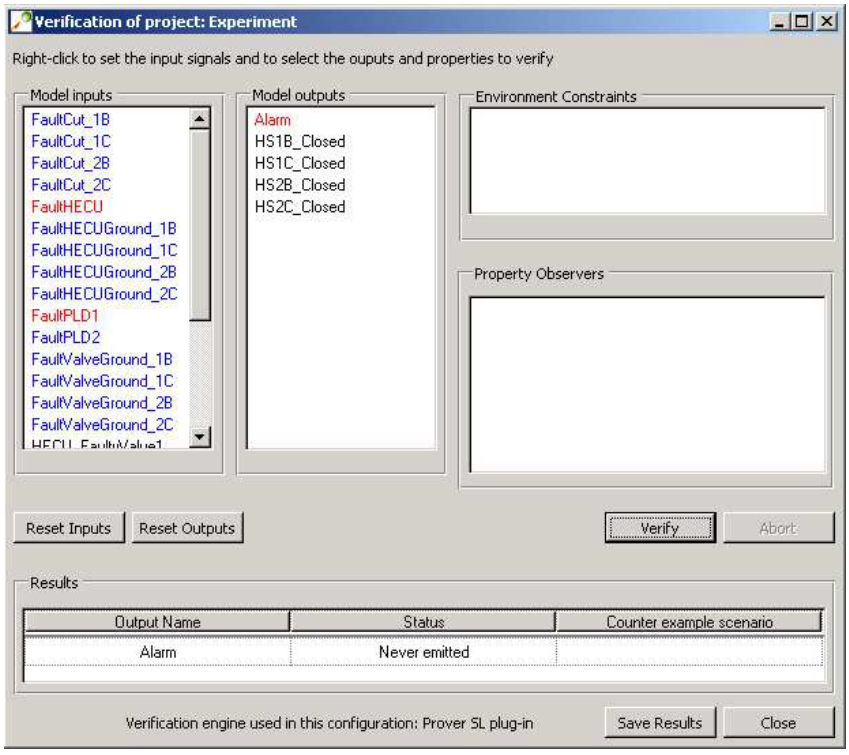

Fig. 9. The model checker window in Esterel Studio.

Figure 9 shows the model checker window in Esterel Studio when verifying the aerospace application verification bench. In the Model Inputs panel, the user can easily constrain the fault mode signals before running the verification. It should be clear that if this model were verified with all fault mode signals

allowed to be present, then the safety properties would most probably be found false, unless the system is extremely robust and can withstand any fault. When a property is found false, a counter-example is produced that shows a sequence of combinations of faults and inputs that lead to safety violation.

Using this technique on the aerospace application verification bench, consisting of 422 lines of code spread of 6 Esterel modules, we could verify the following:

- *The components do not contain design faults causing violation of the property.* This was shown by constraining all fault mode signals to be absent before running the verification.
- *No combination of the twelve valve faults can cause violation of the property.* This was verified by constraining the three component fault mode signals to be absent.
- *No single random fault can cause violation of the property.* The three component fault modes were checked by for each of them constraining the other fault mode signals to be absent. The other twelve faults were already cleared by the previous step.
- *The only double fault violating the property was shown to be when both the H-ECU and PLD2 are faulty.* This was checked by testing the relevant possible double faults, again constraining the other fault mode signals for each pair. Since the four valves are symmetrical, it was sufficient to check physical faults in one valve such as 1B, reducing the number of combinations to twelve.

The model checking never took more than a few seconds, and experiments indicate that SAT-based model checking scales well for real-world industrial systems [19]. This means that systems of this kind totalling thousands of lines of code should still be feasible to verify. Alternatively, if testing for validity takes too long, one may opt for the bug chasing strategy option in Esterel Studio. This option omits the proof-searching part of the model checking and only searches for counter-examples [5], and can thus not be used for proofs, but only for finding fault mode combinations.

5 Conclusions

The conquest for making safe avionic systems while incorporating modern technology and more complex functionality is a driving force for the rising interest in FPGAs in safety-related systems. As with other reusable components, we need guidelines on how to incorporate such components into the

[5] The model checking is an induction proof over the discrete time. The induction depth is increased until either the base step is found false or the inductive step is found true. The bug chasing strategy omits the inductive step.

development and safety analysis processes. Furthermore, we need specific guidelines for treatment of FPGAs when building up safety arguments.

In this paper we provide some evidence that results of the last decade of research in language design, formal verification and tool development are reaching maturity levels that make a serious case for these techniques in real applications. We have illustrated how an FPGA design process can combine analyses for safety and functional correctness, and guide the designer in the search for a focus for fault tolerance methods. The abstract (implementation independent) design model was shown to be transformable to a VHDL implementation with acceptable loss of efficiency (still fitting in the circuit that was intended for the design), and at the same time supporting formal analysis of the design.

We proposed a formal verification bench for analysing systematic and random faults in the external environment, using the standard technique of observers, and showed it to be an efficient means of pinpointing fault combinations that need more attention in safety evaluations. The use of verification benches for safety analyses should be applicable to any design language with formal verification support, not only to Esterel.

Ideally, the analysis should render a set of prime implicants of the system failure function, but this is not possible in the current version of the tool. One algorithmic approach is presented in [2]. Current work on the Esterel verification bench includes extention of the tool so that prime implicants (or FTA-like cut-sets) are automatically generated. Another extention would be visualisations and combination with quantitative methods currently used by engineers.

6 Acknowledgements

This work was supported by the national project SAVE supported by Strategic Research Foundation in Sweden (SSF), and the the national aerospace program NFFP-3-428.

The authors wish to acknowledge the support of contact persons at Saab aerospace, Lars Holmlund, Hans Sjöblom, Anna-Karin Rosen and Marianne Almesåker, and the Hydralic subsystem engineer Thomas Trei for discussions and valuable feedback.

References

[1] Berry, Gerard and Georges Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming **19(2)** (1992), 87-152, Elsevier Science, 1992.

[2] Deneux, Johann, "Automated Fault-Tree analysis", Master's Thesis, Uppsala university, 2001.

[3] Edwards, Stephen A., *High-level Synthesis from the Synchronous Language Esterel*, in Proceedings of the International Workshop of Logic and Synthesis (IWLS), New Orleans, June 2002.

[4] Esterel Technologies web site, URL: `http://www.esterel-technologies.com`

[5] Hammarberg, Jerker, "High-Level Development and Formal Verification of Reconfigurable Hardware", Master's Thesis LiTH-IDA-Ex-02/102, Linköping university, 2002.

[6] Halbwachs, Nicholas, Fabienne Lagnier and Pascal Raymond, *Synchronous Observers and the Verification of Reactive Systems*, Third International Conference on Algebraic Methodology and Software Technology (AMAST93), Workshops in Computing, Springer-Verlag, June 1993.

[7] Halbwachs, Nicholas and Pascal Raymond, *A Tutorial of Lustre*, Technical report, Verimag, September 1993.

[8] Henley, Ernest J. and Hiromitsu Kumamoto, "Reliability Engineering and Risk Assessment", Prentice-Hall, 1981.

[9] Holbrook, Donald, *FPGA Use For Safety Critical Functions in an Air Intercept Missile*, in Proceedings of the 19th International System Safety Conference, pp 618-628, 2001.

[10] INRIA TICK project web page,
URL: `http://www.inria.fr/recherche/equipes/tick.en.html`

[11] Katz, Richard B., *Faster, Better, Cheaper Space Flight Electronics — An Analytical Case Study*, Mil/Aero Applications of Programmable Logic Devices (MAPLD) Conference, September 2000.

[12] Le Guernic, Paul, Thierry Gautier, Michel Le Borgne and Claude Le Maire, *Programming real-time applications with SIGNAL*, in Proceedings of the IEEE, vol 79, pp 1321-1336, September 1991.

[13] Manna, Zohar and Amir Pnueli, "The Temporal Logic of Reactive and Concurrent Systems — Specification", Springer-Verlag, New York, 1992.

[14] McMillan, Kenneth L., "Symbolic Model Checking — An approach to the state explosion problem", Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[15] Musa, John D., Anthony Iannino and Kazuhira Okumoto, "Software Reliability — Measurement, Prediction, Application", McGraw-Hill, 1987.

[16] Shivakumar, Premkishore, Michael Kistler, Stephen W. Keckler, Doug Burger and Lorenzo Alvisi, *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*, in Proceedings of International Conference on Dependable Systems and Networks, pp 389-398, IEEE, June 2002.

[17] Sjöholm, Stefan and Lennart Lindh, "VHDL för konstruktion", Studentlitteratur, Lund, 1999.

[18] Sheeran, Mary, Satnam Singh and Gunnar Stålmarck, *Checking Safety Properties Using Induction and a SAT-Solver*, in Proceedings of Formal Methods in Computer-Aided Design, Springer-Verlag, November 2000.

[19] Sheeran, Mary and Gunnar Stålmarck, *A Tutorial on Stålmarck's Proof Procedure for Propositional Logic*, Formal Methods in System Design **16(1)** (2000), 23-58, Kluwer Academic Publishers, January 2000.

[20] Synplify Pro product web page,
URL: `http://www.synplicity.com/products/synplifypro`

[21] Åkerlund, Ove, Simin Nadjm-Tehrani and Gunnar Stålmarck, *Integration of Formal Methods into System Safety and Reliability Analysis*, in Proceedings of the 17th International System Safety Conference, Orlando, 1999.